Modelling and Analysis of Collective Adaptive Systems with CARMA and its Tools

Modelling and Analysis of Collective Adaptive Systems with CARMA and its Tools

Michele Loreti[1] and Jane Hillston[2]

[1] Dipartimento di Statistica, Informatica, Applicazioni "G. Parenti", Università di Firenze
[2] Laboratory for Foundations of Computer Science, University of Edinburgh

Abstract. Collective Adaptive Systems (CAS) are heterogeneous collections of autonomous task-oriented systems that cooperate on common goals forming a collective system. This class of systems is typically composed of a huge number of interacting agents that dynamically adjust and combine their behaviour to achieve specific goals.
This chapter presents CARMA, a language recently defined to support specification and analysis of collective adaptive systems, and its tools developed for supporting system design and analysis. CARMA is equipped with linguistic constructs specifically developed for modelling and programming systems that can operate in open-ended and unpredictable environments. The chapter also presents the CARMA Eclipse plug-in that allows CARMA models to be specified by means of an appropriate high-level language. Finally, we show how CARMA and its tools can be used to support specification with a simple but illustrative example of a socio-technical collective adaptive system.

1 Introduction

In the last forty years *Process Algebras* (see [3] and the references therein), or *Process Description Languages* (PDL), have been successfully used to model and analyse the behaviour of concurrent and distributed systems. A Process Algebra is a formal language, equipped with a rigorous semantics, that provides models in terms of processes. These are agents that perform actions and communicate (interact) with similar agents and with their environment.

At the beginning, Process Algebras were only focussed on *qualitative aspects* of computations. However, when complex and large-scale systems are considered, it may not be sufficient to check if a property *is satisfied or not*. This is because random phenomena are a crucial part of *distributed systems* and one is also interested in verifying *quantitative aspects* of computations.

This motivated the definition of a new class of PDL where *time* and *probabilities* are explicitly considered. This new family of formalisms have proven to be particularly suitable for capturing important properties related to performance and quality of service, and even for the modelling of biological systems. Among others we can refer here to PEPA [19], MTIPP [18], EMPA [4], Stochastic π-Calculus [23], Bio-PEPA [9], MODEST [5] and others [17, 8].

The ever increasing complexity of systems has further changed the perspective of the system designer that now has to consider a new class of systems, named *Collective adaptive systems* (CAS), that consist of massive numbers of components, featuring

complex interactions among components and with humans and other systems. Each component in the system may exhibit autonomic behaviour depending on its properties, objectives and actions. Decision-making in such systems is complicated and interaction between their components may introduce new and sometimes unexpected behaviours.

CAS operate in open and non-deterministic environments. Components may enter or leave the collective at any time. Components can be highly heterogeneous (machines, humans, networks, etc.) each operating at different temporal and spatial scales, and having different (potentially conflicting) objectives.

CAS thus provide a significant research challenge in terms of both representation and reasoning about their behaviour. The pervasive yet transparent nature of the applications developed in this paradigm makes it of paramount importance that their behaviour can be thoroughly assessed during their design, prior to deployment, and throughout their lifetime. Indeed their adaptive nature makes modelling essential and models play a central role in driving their adaptation. Moreover, the analysis should encompass both functional and non-functional aspects of behaviour. Thus it is vital that we have available robust modelling techniques which are able to describe such systems and to reason about their behaviour in both qualitative and quantitative terms. To move towards this goal, it is important to develop a theoretical foundation for CAS that will help in understanding their distinctive features. From the point of view of the language designers, the challenge is to devise appropriate abstractions and linguistic primitives to deal with the large dimension of systems, to guarantee adaptation to (possibly unpredicted) changes of the working environment, to take into account evolving requirements, and to control the emergent behaviours resulting from complex interactions.

To design this new language for CAS we first have identified the *design principles* together with the *primitives* and *interaction patterns* that are needed in CAS design. Emphasis has been given placed on identifying the appropriate abstractions and linguistic primitives for modelling and programming collective adaptation, locality representation, knowledge handling, and system interaction and aggregation.

To be effective, any language for CAS should provide:

- Separation of knowledge and behaviour;
- Control over abstraction levels;
- Bottom-up design;
- Mechanisms to take into account the environment;
- Support for both global and local views; and
- Automatic derivation of the underlying mathematical model.

These design principles have been the starting point for the design of a language, developed specifically to support the specification and analysis of CAS, with the particular objective of supporting quantitative evaluation and verification. We named this language CARMA, Collective Adaptive Resource-sharing Markovian Agents [7, 20].

CARMA combines the lessons which have been learned from the long tradition of stochastic process algebras, with those more recently acquired from developing languages to model CAS, such as SCEL [12] and PALOMA [13], which feature attribute-based communication and explicit representation of locations.

SCEL [12] (Software Component Ensemble Language), is a kernel language that has been designed to support the programming of autonomic computing systems. This

language relies on the notions of *autonomic components* representing the collective members, and *autonomic-component ensembles* representing collectives. Each component is equipped with an interface, consisting of a collection of attributes, describing different features of components. Attributes are used by components to dynamically organise themselves into ensembles and as a means to select partners for interaction. The stochastic variant of SCEL, called StocS [22], was a first step towards the investigation of the impact of different stochastic semantics for autonomic processes, that relies on stochastic output semantics, probabilistic input semantics and on a probabilistic notion of knowledge. Moreover, SCEL has inspired the development of the core calculus AbC [2, 1] that focuses on a minimal set of primitives that defines attribute-based communication, and investigates their impact. Communication among components takes place in a broadcast fashion, with the characteristic that only components satisfying predicates over specific attributes receive the sent messages, provided that they are willing to do so.

PALOMA [13] is a process algebra that takes as its starting point a model based on located Markovian agents each of which is parameterised by a location, which can be regarded as an attribute of the agent. The ability of agents to communicate depends on their location, through a perception function. This can be regarded as an example of a more general class of attribute-based communication mechanisms. The communication is based on a multicast, as only agents who enable the appropriate reception action have the ability to receive the message. The scope of communication is thus adjusted according to the perception function.

A distinctive contribution of the language CARMA is the rich set of communication primitives that are offered. This new language supports both unicast and broadcast communication, and locally synchronous, but globally asynchronous communication. This richness is important to enable the spatially distributed nature of CAS, where agents may have only local awareness of the system, yet the design objectives and adaptation goals are often expressed in terms of global behaviour. Representing these rich patterns of communication in classical process algebras or traditional stochastic process algebras would be difficult, and would require the introduction of additional model components to represent buffers, queues, and other communication structures. Another feature of CARMA is the explicit representation of the environment in which processes interact, allowing rapid testing of a system under different open world scenarios. The environment in CARMA models can evolve at runtime, due to the feedback from the system, and it further modulates the interaction between components, by shaping rates and interaction probabilities.

The focus of this tutorial is the presentation of the language and its discrete semantics, which are presented in the FUTS style [11]. The structure of the chapter is as follows. Section 2 presents the syntax of the language and explains the organisation of a model in terms of a collective of agents that are considered in the context of an environment. In Section 3 we give a detailed account of the semantics, particularly explaining the role of the environment. The use of CARMA is illustrated in Section 4 where we describe a model of a simple bike sharing system, and explain the support given to the CARMA modeller in the current implementation. Section 5 considers the bike sharing

system in different scenarios, demonstrating the analytic power of the CARMA tools. Some conclusions are drawn in Section 6.

2 CARMA: Collective adaptive resource-sharing Markovian agents

CARMA is a new stochastic process algebra for the representation of systems developed according to the CAS paradigm [7, 20]. The language offers a rich set of communication primitives, and the exploitation of attributes, captured in a store associated with each component, to enable attribute-based communication. For most CAS systems we anticipate that one of the attributes could be the location of the agent [15]. Thus it is straightforward to model those systems in which, for example, there is limited scope of communication or, restriction to only interact with components that are co-located, or where there is spatial heterogeneity in the behaviour of agents.

The rich set of communication primitives is one of the distinctive features of CARMA. Specifically, CARMA supports both unicast and broadcast communication, and permits locally synchronous, but globally asynchronous communication. This richness is important to take into account the spatially distributed nature of CAS, where agents may have only local awareness of the system, yet the design objectives and adaptation goals are often expressed in terms of global behaviour. Representing these patterns of communication in classical process algebras or traditional stochastic process algebras would be difficult, and would require the introduction of additional model components to represent buffers, queues and other communication structures.

Another key feature of CARMA is its distinct treatment of the *environment*. It should be stressed that although this is an entity explicitly introduced within our models, it is intended to represent something more pervasive and diffusive of the real system, which is abstracted within the modelling to be an entity which exercises influence and imposes constraints on the different agents in the system. For example, in a model of a smart transport system, the environment may have responsibility for determining the rate at which entities (buses, bikes, taxis etc) move through the city. However this should be recognised as an abstraction of the presence of other vehicles causing congestion which may impede the progress of the focus entities to a greater or lesser extent at different times of the day. The presence of an environment in the model does not imply the existence of centralised control in the system. The role of the environment is also related to the spatially distributed nature of CAS — we expect that the location *where* an agent is will have an effect on *what* an agent can do.

This view of the environment coincides with the view taken by many researchers within the situated multi-agent community e.g. [26]. Specifically, in [27] Weyns *et al.* argue about the importance of having a distinct environment within every multi-agent system. Whilst they are viewing such systems from the perspective of software engineers, many of their arguments are as valid when it comes to modelling a multi-agent or collective adaptive system. Thus our work can be viewed as broadly fitting within the same framework, albeit with a higher level of abstraction. Just as in the construction of a system, in the construction development of a model distinguishing clearly between the responsibilities of the agents and of the environment provides separation of concerns and assists in the management of complex systems.

In [27] the authors provide the following definition: "*The environment is a first-class abstraction that proves the surrounding conditions for agents to exist and that mediates both the interaction among agents and the access to resources.*" This is the role that the environment plays within CARMA models through the evolution rules. However, in contrast to the framework of Weyns *et al.*, the environment in a CARMA model is not an active entity in the same sense as the agents are active entities. In our case, the environment is constrained to work *through* the agents, by influencing their dynamic behaviour or by inducing changes in the number and types of agents making up the system.

In [24], Saunier *et al*. advocate the use of an *active environment* to mediate the interactions between agents; such an active environment is aware of the current context for each agent. The environment in CARMA also supports this view, as the evolution rules in the environment take into account the state of all the potentially participating components to determine both the rate and the probability of communications being successful, thus achieving a multicast communication not based on the address of the receiving agents, as suggested by Saunier *et al*. This is what we term "attribute-based communication" in CARMA. Moreover, when the application calls for a centralised information portal, the global store in CARMA can represent it. The higher level of abstraction offered by CARMA means that many implementation issues are ignored.

2.1 A running example

To describe basic features of CARMA a *running example* will be used. This is based on a *bike sharing system* (BSS) [10]. These systems are a recent, and increasingly popular, form of public transport in urban areas. As a resource-sharing system with large numbers of independent users altering their behaviour due to pricing and other incentives [14], they are a simple instance of a collective adaptive system, and hence a suitable case study to exemplify the CARMA language.

The idea in a bike sharing system is that bikes are made available in a number of stations that are placed in various areas of a city. Users that plan to use a bike for a short trip can pick up a bike at a suitable origin station and return it to any other station close to their planned destination. One of the major issues in bike sharing systems is the availability and distribution of resources, both in terms of available bikes at the stations and in terms of available empty parking places in the stations.

In our scenario we assume that the city is partitioned in homogeneous zones and that all the *stations* in the same zone can be equivalently used by any user in that zone. Below, we let $\{z_0, \ldots, z_n\}$ be the n zones in the city, each of which contains k parking stations.

2.2 A gentle introduction to CARMA

The bike sharing systems described in the previous section represent well typical scenarios that can be modelled with CARMA. Indeed, a CARMA system consists of a *collective* (N) operating in an *environment* ($\mathscr{E}$). The collective is a multiset of components that models the behavioural part of a system; it is used to describe a group of interacting *agents*. The environment models all those aspects which are intrinsic to the context

where the agents under consideration are operating. The environment also mediates agent interactions.

Example 1. Bike Sharing System (1/7) In our running example the collective N will be used to model the behaviour of *parking stations* and *users*, while the environment will be used to model the city context where these agents operate like, for instance, the user arrival rate or the possible destinations of trips. □

We let SYS be the set of CARMA *systems* S defined by the following syntax:

$$S ::= N \textbf{ in } \mathscr{E}$$

where is a collective and is an environment.

Collectives and Components. We let COL be the set of collectives N which are generated by the following grammar:

$$N ::= C \mid N \parallel N$$

A collective N is either a *component* C or the parallel composition of collectives $N_1 \parallel N_2$. The former identifies a multiset containing the single component C while the latter represents the union of the multisets denoted by N_1 and N_2, respectively. In the rest of this chapter we will sometimes use standard operations on multisets over a collective. We use $N(C)$ to indicate the multiplicity of C in N, $C \in N$ to indicate that $N(C) > 0$ and $N - C$ to represent the collective obtained from N by removing component C.

The precise syntax of components is:

$$C ::= \mathbf{0} \mid (P, \gamma)$$

where we let COMP be the set of components C generated by the previous grammar.

A component C can be either the *inactive component*, which is denoted by $\mathbf{0}$, or a term of the form (P, γ), where P is a *process* and γ is a *store*. A term (P, γ) models an *agent* operating in the system under consideration: the process P represents the agent's behaviour whereas the store γ models its *knowledge*. A store is a function which maps *attribute names* to *basic values*. We let:

- ATTR be the set of *attribute names* $a, a', a_1, \ldots, b, b', b_1, \ldots$;
- VAL be the set of *basic values* $v, v', v_1, \ldots$;
- Γ be the set of *stores* $\gamma, \gamma_1, \gamma', \ldots$, i.e. functions from ATTR to VAL.

Example 2. Bike Sharing System (2/7) To model our *Bike Sharing System* in CARMA we need two kinds of components, one for each of the two groups of agents involved in the system, i.e. *parking stations* and *users*. Both kinds of component use the local store to publish the relevant data that will be used to represent the state of the agent. We can notice that, following this approach, bikes are not explicitly modelled in the system. This is because we are interested in modelling only the behaviour of the *active* components in the system. Under this perspective, bikes are just the resources exchanged by *parking stations* and *users*.

The local store of components associated with *parking stations* contains the following attributes:

– loc: identifying the zone where the parking station is located;
– capacity: describing the maximal number of parking slots available in the station;
– available: indicating the current number of bikes currently available in the parking station.

Similarly, the local store of components associated with *users* contains the following attributes:

– loc: indicating current user location;
– dest: indicating user destination.

□

Processes. The behaviour of a component is specified via a process P. We let PROC be the set of CARMA processes $P, Q, \ldots$ defined by the following grammar:

$$
\begin{array}{lll}
P,Q ::= \mathbf{nil} & & act ::= \alpha^\star[\pi_s]\langle\overrightarrow{e}\rangle\sigma \\
\quad \mid act.P & & \quad \mid \alpha[\pi_r]\langle\overrightarrow{e}\rangle\sigma \\
\quad \mid P+Q & & \quad \mid \alpha^\star[\pi_s](\overrightarrow{x})\sigma \\
\quad \mid P \mid Q & & \quad \mid \alpha[\pi_r](\overrightarrow{x})\sigma \\
\quad \mid [\pi]P & & \\
\quad \mid \mathbf{kill} & & e ::= a \mid \mathsf{my}.a \mid x \mid v \mid \mathsf{now} \mid \cdots \\
\quad \mid A \qquad (A \triangleq P) & & \pi_s, \pi_r, \pi ::= \top \mid \bot \mid e_1 \bowtie e_2 \mid \neg\pi \mid \pi \wedge \pi \mid \cdots
\end{array}
$$

Above, the following notation is used:

– α is an *action type* in the set ACTTYPE;
– π is a *predicate*;
– x is a *variable* in the set of variables VAR;
– e is an expression in the set of expressions EXP[3];
– $\overrightarrow{\cdot}$ indicates a sequence of elements;
– σ is an *update*, i.e. a function from Γ to $Dist(\Gamma)$ in the set of *updates* Σ; where $Dist(\Gamma)$ is the set of probability distributions over Γ.

CARMA processes are built by using standard operators of process algebras. Basic processes can be either **nil** or **kill**. The former represents the *inactive process* while the latter is used, when activated, to *destroy* a component. We assume that the term **kill** always occurs under the scope of an action prefix.

Choice $(\cdot + \cdot)$ and parallel composition $(\cdot|\cdot)$ are the usual process algebra operators: $P_1 + P_2$ indicates a process that can behave either like P_1 or like P_2; while the behaviour of $P_1|P_2$ is the result of the interleaving between P_1 and P_2. In the next section, when the stochastic operational semantics of CARMA will be presented, we will show how possible alternative computations of a process P are probabilistically selected.

[3] The precise syntax of expressions e has been deliberately omitted. We only assume that expressions are built using the appropriate combinations of *values*, *attributes* (sometime prefixed with my), variables and the special term now. The latter is used to refer to current time unit.

Process behaviour can be influenced by the store γ of the hosting component. This is the case of the *guard* operator $[\pi]P$ where the process P is activated when the predicate π, i.e. a boolean expression over *attribute names*, is satisfied (otherwise it is inactive). This operator can be used to enable a given behaviour only when some conditions are satisfied. In the case of our *Bike Sharing System*, if P_c is the behaviour modelling *bike retrieval*, a prediate of the form $\mathsf{available} > 0$ can be used to enable P_c only when there are bikes available.

CARMA processes located in different components interact while performing four types of actions: *broadcast output* ($\alpha^\star[\pi]\langle\overrightarrow{e}\rangle\sigma$), *broadcast input* ($\alpha^\star[\pi](\overrightarrow{x})\sigma$), *output* ($\alpha[\pi]\langle\overrightarrow{e}\rangle\sigma$), and *input* ($\alpha[\pi](\overrightarrow{x})\sigma$),

The admissible communication partners of each of these actions are identified by the predicate π. Note that, in a component (P,γ) the store γ regulates the behaviour of P. Primarily, γ is used to evaluate the predicate associated with an action in order to filter the possible synchronisations involving process P. In addition, γ is also used as one of the parameters for computing the actual rate of actions performed by P. The process P can change γ immediately after the execution of an action. This change is brought about by the *update* σ. The update is a function that when given a store γ returns a probability distribution over Γ which expresses the possible evolutions of the store after the action execution.

The *broadcast output* $\alpha^\star[\pi]\langle\overrightarrow{e}\rangle\sigma$ models the execution of an action α that spreads the values resulting from the evaluation of expressions $\overrightarrow{e}$ in the local store γ. This message can be potentially received by any process located at components whose store satisfies predicate π. This predicate may contain references to attribute names that have to be evaluated under the local store. For instance, if loc is the attribute used to store the position of a component, action

$$\alpha^\star[\mathsf{my.loc} == \mathsf{loc}]\langle\overrightarrow{v}\rangle\sigma$$

potentially involves all the components located at the same location. The *broadcast output* is non-blocking. The action is executed even if no process is able to receive the values which are sent. Immediately after the execution of an action, the update σ is used to compute the (possible) *effects* of the performed action on the store of the hosting component where the output is performed.

To receive a broadcast message, a process executes a *broadcast input* of the form $\alpha^\star[\pi](\overrightarrow{x})\sigma$. This action is used to receive a tuple of values $\overrightarrow{v}$ sent with an action α from a component whose store satisfies the predicate $\pi[\overrightarrow{v}/\overrightarrow{x}]$. The transmitted values can be part of the predicate π. For instance, $\alpha^\star[x > 5](x)\sigma$ can be used to receive a value that is greater than 5.

The other two kinds of action, namely *output* and *input*, are similar. However, differently from broadcasts described above, these actions realise a *point-to-point* interaction. The *output* operation is blocking, in contrast with the non-blocking broadcast output.

Example 3. Bike Sharing System (3/7) We are now ready to describe the behaviour of *parking stations* and *users* components.

Each parking station is modelled in CARMA via a component of the form:

$$(\ G|R\ ,\{\mathsf{loc} = \ell, \mathsf{capacity} = i, \mathsf{available} = j\})$$

where loc is the attribute that identifies the zone where the parking station is located; capacity indicates the number of parking slots available in the station; available is the number of available bikes.

Processes G and R, which model the procedure to *get* and *return* a bike in the parking station, respectively, are defined as follows:

$$G \triangleq [\mathsf{available} > 0]\ \mathsf{get}[\mathsf{my.loc} == \mathsf{loc}]\langle\bullet\rangle\{\mathsf{available} \leftarrow \mathsf{available} - 1\}.G$$

$$R \triangleq [\mathsf{available} < \mathsf{capacity}]\ \mathsf{ret}[\mathsf{my.loc} == \mathsf{loc}]\langle\bullet\rangle\{\mathsf{available} \leftarrow \mathsf{available} + 1\}.R$$

When the value of attribute available is greater than 0, process G executes the *unicast output* with action type get that potentially involves components satisfying the predicate my.loc $==$ loc, i.e. the ones that are located in the same zone[4]. When the output is executed the value of the attribute available is decreased by one to model the fact that one bike has been retrieved from the parking station.

Process R is similar. It executes the *unicast output* with action type ret that potentially involves components satisfying predicate my.loc $==$ loc. This action can be executed only when there is at least one parking slot available, i.e. when the value of attribute available is less than the value of attribute capacity. When the output considered above is executed, the value of attribute available is increased by one to model the fact that one bike has been returned in the parking station.

Users, who can be either *bikers* or *pedestrians*, are modelled via components of the form:

$$(Q, \{\mathsf{loc} = \ell_1, \mathsf{dest} = \ell_2\})$$

where loc is the attribute indicating where the user is located, while dest indicates the user destination. Process Q models the current state of the user and can be one of the following processes:

$$P \triangleq \mathsf{get}[\mathsf{my.loc} == \mathsf{loc}](\bullet).B$$

$$B \triangleq \mathsf{move}^{\star}[\bot]\langle\bullet\rangle\{\mathsf{loc} \leftarrow \mathsf{dest}\}.W$$

$$W \triangleq \mathsf{ret}[\mathsf{my.loc} == \mathsf{loc}](\bullet).\mathbf{kill}$$

Process P represents a *pedestrian*, i.e. a user that is waiting for a bike. To *get a bike* a *pedestrian* executes a *unicast input* over activity get while selecting only parking stations that are located in his/her current location (my.loc $==$ loc). When this action is executed, a *pedestrian* becomes a *biker* B.

A biker can *move* from the current zone to the destination. This activity is modelled with the execution of a broadcast output via action type move. Note that, the predicate used to identify the target of the actions is $\bot$, denoting the value *false*. This means that this action actually does not synchronise with any component (since $\bot$ is never satisfied). This kind of *pattern* is used in CARMA to model *spontaneous actions*, i.e. actions that render the execution of an activity and that do not require synchronisation.

[4] Here we use $\bullet$ to denote the unit value.

After the broadcast $\mathsf{move}^\star$ the value of attribute loc is updated to dest and process W is activated. We will see in the next section that the actual rate of this action is determined by the environment and may also depend on the current time.

Process W represents a user who is waiting for a parking slot. This process executes an input over ret. This models the fact that the user has found a parking station with an available parking slot in their zone. After the execution of this input the user *disappears* from the system since the process **kill** is activated.

To model the arrival of new users, the following component is used:

$$(A, \{\mathsf{loc} = \ell\})$$

where attribute loc indicates the location where users arrive, while process A is:

$$A \stackrel{\triangle}{=} \mathsf{arrival}^\star[\bot]\langle\bullet\rangle\{\}.A$$

This process only performs the spontaneous action $\mathsf{arrival}$. The precise role of this process will be clear in a few paragraphs when the environment will be described. □

Environment. An environment consists of two elements: a *global store* γ_g, that models the overall state of the system, and an *evolution rule* ρ.

Example 4. Bike Sharing System (4/7) The global store can be used to describe global information that may affect the system behaviour. In our *Bike Sharing System* we use the attribute user to record the number of active users.

The *evolution rule* ρ is a function which, depending on the *current time*, on the global store and on the current state of the collective (i.e., on the configurations of each component in the collective) returns a tuple of functions $\varepsilon = \langle \mu_p, \mu_w, \mu_r, \mu_u \rangle$ known as the *evaluation context* where $\text{ACT} = \text{ACTTYPE} \cup \{\alpha^\star | \alpha \in \text{ACTTYPE}\}$ and:

- $\mu_p : \Gamma \times \Gamma \times \text{ACT} \to [0,1]$, $\mu_p(\gamma_s, \gamma_r, \alpha)$ expresses the probability that a component with store γ_r can receive a broadcast message from a component with store γ_s when α is executed;
- $\mu_w : \Gamma \times \Gamma \times \text{ACT} \to [0,1]$, $\mu_w(\gamma_s, \gamma_r, \alpha)$ yields the weight will be used to compute the probability that a component with store γ_r can receive a unicast message from a component with store γ_s when α is executed;
- $\mu_r : \Gamma \times \text{ACT} \to \mathbb{R}_{\geq 0}$, $\mu_r(\gamma_s, \alpha)$ computes the execution rate of action α executed at a component with store γ_s;
- $\mu_u : \Gamma \times \text{ACT} \to \Sigma \times \text{COL}$, $\mu_u(\gamma_s, \alpha)$ determines the updates on the environment (global store and collective) induced by the execution of action α at a component with store γ_s.

For instance, the probability to receive a given message may depend on the *number or faction* of components in a given state. Similarly, the actual rate of an action may be a function of the number of components whose store satisfies a given property.

Functions μ_p and μ_w play a similar role. However, while the former computes the probability that a component receives a broadcast message, the latter associates to each

unicast interaction with a weight, i.e. a non negative real number. This weight will be used to compute the probability that a given component with store γ_r receives a unicast message over activity α from a component with store γ_r. This probability is obtained by dividing the weight $\mu_w(\gamma_s, \gamma_r, \alpha)$ by the *total weights* of all possible receivers.

Example 5. Bike Sharing System (5/7) In our scenario, function μ_w can have the following form:

$$\mu_w(\gamma_s, \gamma_r, \alpha) = \begin{cases} 1 & \alpha = \mathsf{get} \wedge \gamma_s(\mathsf{loc}) = \gamma_r(\mathsf{loc}) \\ 1 & \alpha = \mathsf{ret} \wedge \gamma_s(\mathsf{loc}) = \gamma_r(\mathsf{loc}) \\ 0 & \text{otherwise} \end{cases}$$

where γ_s is the store of the sender, γ_r is the store of the receiver. The above function imposes that all the users in the same zone have the same weight, that is 1 when a user is located in the same zone of the parking station and 0 otherwise. This means that each user in the same zone have the same probability to be selected for getting a bike or for using a parking slot at a station. The weight associated to all the other interactions is 0. □

Function μ_r computes the rate of a unicast/broadcast output. This function takes as parameter the local store of the component performing the action and the action on which the interaction is based. Note that the environment can disable the execution of a given action. This happens when the function μ_r (resp. μ_p or μ_w) returns the value 0.

Example 6. Bike Sharing System (6/7) In our example μ_r can be defined as follows:

$$\mu_r(\gamma_s, \alpha) = \begin{cases} \lambda_g & \alpha = \mathsf{get} \\ \lambda_r & \alpha = \mathsf{ret} \\ \mathit{mtime}(\mathsf{now}, \gamma_s(\mathsf{loc}), \gamma_s(\mathsf{dest})) & \alpha = \mathsf{move}^\star \\ \mathit{atime}(\mathsf{now}, \gamma_s(\mathsf{loc}), \gamma_g(\mathsf{users})) & \alpha = \mathsf{arrival}^\star \\ 0 & \text{otherwise} \end{cases}$$

We say that actions get and ret are executed at a constant rate; the rate of movement is a function (*mtime*) of actual time (now) and of starting location and final destination. Rate of user arrivals (computed by function *atime*) depends on current time now on location loc and on the number of users that are currently active in the system[5]. All the other interactions occurs with rate 0. □

Finally, the function μ_u is used to update the global store and to activate a new collective in the system. The function μ_u takes as parameters the store of the component performing the action together with the action type and returns a pair (σ, N). Within this pair, σ identifies the update on the global store whereas N is a new collective installed in the system. This function is particularly useful for modelling the arrival of new agents into a system.

[5] Here we assume that functions *mtime* and *atime* are obtained after some observations on real systems

Example 7. Bike Sharing System (7/7) In our scenario function update is used to model the arrival of new users and it is defined as follows:

$$\mu_u(\gamma_s,\alpha)=\begin{cases}\{\mathsf{users}\leftarrow\gamma_g(\mathsf{users})+1\},\\ \qquad (W,\{\mathsf{loc}=\gamma_s(\mathsf{loc}),\mathsf{dest}=\mathit{destLoc}(\mathsf{now},\gamma_s(\mathsf{loc}))\}) & \alpha=\mathsf{arrival}^\star\\ \{\mathsf{users}\leftarrow\gamma_g(\mathsf{users})-1\},0 & \alpha=\mathsf{ret}\\ \{\},0 & \text{otherwise}\end{cases}$$

When action $\mathsf{arrival}^\star$ is performed a component associated with a new user is created in the same location as the sender (see Example 3). The destination of the new user will be determined by function *destLoc* that takes the current system time and starting location and returns a probability distribution over locations. Moreover, the global store records that a new user entered in the system. The number of active users is decremented by 1 each time action ret is performed. All the other actions do not trigger any update on the environment. □

3 CARMA semantics

The operational semantics of CARMA specifications is defined in terms of three functions that compute the possible *next states* of a *component*, a *collective* and a *system*:

1. the function $\mathbb{C}$ that describes the behaviour of a single component;
2. the function $\mathbb{N}_\varepsilon$ builds on $\mathbb{C}$ to describe the behaviour of collectives;
3. the function $\mathbb{S}_t$ that shows how CARMA systems evolve.

Note that, classically behaviour of (stochastic) process algebras is represented via *transition relations*. These relations, defined following a Plotkin-style, are used to infer possible computations of a process. Note that, due to *nondeterminism*, starting from the same process, different evolutions can be inferred. However, in CARMA, there is not any form of nonterminism while the selection of possible next state is governed by a probability distribution.

In this chapter we use an approach based on FUTS style [11]. Using this approach, the behaviour of a term is described using a function that, given a *term* and a *transition label*, yields a function associating each component, collective, or system with a non-negative number. The meaning of this value depends on the context. It can be the rate of the exponential distribution characterising the time needed for the execution of the action represented by ℓ; the probability of receiving a given broadcast message or the weight used to compute the probability that a given component is selected for the synchronisation. In all the cases the zero value is associated with unreachable terms.

We use the FUTS style semantics because it makes explicit an underlying (time-inhomogeneous) Action Labelled Markov Chain, which can be simulated with standard algorithms [16] but is nevertheless more compact than Plotkin-style semantics, as the functional form allows different possible outcomes to be treated within a single rule. A complete description of FUTS and their use can be found in [11].

$$\frac{}{\mathbb{C}[(\mathbf{nil},\gamma),\ell]=\emptyset}\ \textbf{Nil} \qquad \frac{}{\mathbb{C}[\mathbf{0},\ell]=\emptyset}\ \textbf{Zero}$$

$$\frac{[\![\pi_s]\!]_\gamma=\pi_s' \quad [\![\overrightarrow{e}]\!]_\gamma=\overrightarrow{v} \quad \mathbf{p}=\sigma(\gamma)}{\mathbb{C}[(\alpha^\star[\pi_s]\langle\overrightarrow{e}\rangle\sigma.P,\gamma),\alpha^\star[\pi_s']\langle\overrightarrow{v}\rangle,\gamma]=(P,\mathbf{p})}\ \textbf{B-Out}$$

$$\frac{[\![\pi_s]\!]_\gamma=\pi_s' \quad [\![\overrightarrow{e}]\!]_\gamma=\overrightarrow{v} \quad \ell\neq\alpha^\star[\pi_s']\langle\overrightarrow{v}\rangle,\gamma}{\mathbb{C}[(\alpha^\star[\pi_s]\langle\overrightarrow{e}\rangle\sigma.P,\gamma),\ell]=\emptyset}\ \textbf{B-Out-F1}$$

$$\frac{\gamma_r\models\pi_s \quad \gamma_s\models\pi_r[\overrightarrow{v}/\overrightarrow{x}] \quad \mathbf{p}=\sigma[\overrightarrow{v}/\overrightarrow{x}](\gamma_2)}{\mathbb{C}[(\alpha^\star[\pi_r](\overrightarrow{x})\sigma.P,\gamma_r),\alpha^\star[\pi_s](\overrightarrow{v}),\gamma_s]=(P[\overrightarrow{v}/\overrightarrow{x}],\mathbf{p})}\ \textbf{B-In}$$

$$\frac{\gamma_r\not\models\pi_s\vee\gamma_s\not\models\pi_r[\overrightarrow{v}/\overrightarrow{x}]}{\mathbb{C}[(\alpha^\star[\pi_r](\overrightarrow{x})\sigma.P,\gamma_r),\alpha^\star[\pi_s](\overrightarrow{v}),\gamma_s]=\emptyset}\ \textbf{B-In-F1}$$

$$\frac{\ell\neq\alpha^\star[\pi_s](\overrightarrow{v}),\gamma_s}{\mathbb{C}[(\alpha^\star[\pi_r](\overrightarrow{x})\sigma.P,\gamma_r),\ell]=\emptyset}\ \textbf{B-In-F2}$$

Table 1: Operational semantics of components (Part 1)

3.1 Operational semantics of components

The behaviour of a single component is defined by a function

$$\mathbb{C}:\textsc{Comp}\times\textsc{Lab}\to[\textsc{Comp}\to\mathbb{R}_{\geq 0}]$$

Function $\mathbb{C}$ takes a component and a transition label, and yields a function in $[\textsc{Comp}\to\mathbb{R}_{\geq 0}]$. LAB is the set of transition labels ℓ which are generated by the following grammar, where π_s is defined in Section 2.2:

$$\begin{aligned}\ell ::= {} & \alpha^\star[\pi_s]\langle\overrightarrow{v}\rangle,\gamma \text{ Broadcast Output}\\ \mid {} & \alpha^\star[\pi_s](\overrightarrow{v}),\gamma \text{ Broadcast Input}\\ \mid {} & \alpha[\pi_s]\langle\overrightarrow{v}\rangle,\gamma \text{ Unicast Output}\\ \mid {} & \alpha[\pi_s](\overrightarrow{v}),\gamma \text{ Unicast Input}\end{aligned}$$

These labels are associated with the four CARMA input-output actions and contain a reference to the action which is performed (α or $\alpha^\star$), the predicate π_s used to identify the target of the actions, and the value which is transmitted or received.

Function $\mathbb{C}$ is formally defined in Table 1 and Table 2 and shows how a single component evolves when a *input/output* action is executed. For any component C and transition label ℓ, $\mathbb{C}[C,\ell]$ indicates the possible next states of C after the transition ℓ. These states are weighted. If $\mathbb{C}[C,\ell]=\mathscr{C}$ and $\mathscr{C}(C')=p$ then C evolves to C' with a weight p when ℓ is executed.

The process **nil** denotes the process that cannot perform any action. The behaviour associated to this process at the level of components can be derived via the rule **Nil**. This rule states that the inactive process cannot perform any action. This is derived from the fact that function $\mathbb{C}$ maps any label to function $\emptyset$ (rule **Nil**), where $\emptyset$ denotes the 0 constant function.

The behaviour of a *broadcast output* $(\alpha^\star[\pi_s]\langle\overrightarrow{e}\rangle\sigma.P,\gamma)$ is described by rules **B-Out** and **B-Out-F1**. Rule **B-Out** states that a broadcast output $\alpha^\star[\pi_s]\langle\overrightarrow{e}\rangle\sigma$ sends message $[\![\overrightarrow{e}]\!]_\gamma$[6] to all components that satisfy $[\![\pi_s]\!]_\gamma = \pi'_s$. The possible next local stores after the execution of an action are determined by the update σ. This takes the store γ and yields a probability distribution $\mathbf{p} = \sigma(\gamma) \in \mathit{Dist}(\Gamma)$. In rule **B-Out**, and in the rest of the chapter, the following notations are used:

- let $P \in \textsc{Proc}$ and $\mathbf{p} \in \mathit{Dist}(\Gamma)$, $(P,\mathbf{p})$ is a probability distribution in $\mathit{Dist}(\textsc{Comp})$ such that:
$$(P,\mathbf{p})(C) = \begin{cases} 1 & P \equiv Q|\mathbf{kill} \wedge C \equiv \mathbf{0} \\ \mathbf{p}(\gamma) & C \equiv (P,\gamma) \wedge P \not\equiv Q|\mathbf{kill} \\ 0 & \text{otherwise} \end{cases}$$
- let $\mathbf{c} \in \mathit{Dist}(\textsc{Comp})$ and $r \in \mathbb{R}_{\geq 0}$, $r \cdot \mathbf{c}$ denotes the function $\mathscr{C} : \textsc{Comp} \to \mathbb{R}_{\geq 0}$ such that: $\mathscr{C}(C) = r \cdot \mathbf{c}(C)$

Note that, after the execution of an action a component can be destroyed. This happens when the continuation process after the action prefix contains the term **kill**. For instance, by applying rule **B-Out** we have that:

$$\mathbb{C}[(\alpha^\star[\pi_s]\langle v\rangle\sigma.(\mathbf{kill}|Q),\gamma),\alpha^\star[\pi_s]\langle v\rangle,\gamma] = [\mathbf{0} \mapsto r]$$

Rule **B-Out-F1** states that a *broadcast output* can be only involved in labels of the form $\alpha^\star[\pi_s]\langle\overrightarrow{v}\rangle,\gamma$.

Computations related to a broadcast input are labelled with $\alpha^\star[\pi_s](\overrightarrow{v}),\gamma_1$. There, π_s is the predicate used by the sender to identify the target components while $\overrightarrow{v}$ is the sequence of transmitted values. Rule **B-In** states that a component $(\alpha^\star[\pi_r](\overrightarrow{x})\sigma.P,\gamma_r)$ can evolve with this label when its store γ_r (the store of the receiver) satisfies the sender predicate, i.e. $\gamma_r \models \pi_s$, while the store of the sender, i.e. γ_s satisfies the predicate of the receiver $\pi_r[\overrightarrow{v}/\overrightarrow{x}]$.

Rule **B-In-F1** models the fact that if a component is not in the set of possible receivers ($\gamma_r \not\models \pi_s$) or the received values do not satisfy the expected requirements then the component cannot receive a broadcast message. Finally, the rule **B-In-F2** models the fact that $(\alpha^\star[\pi_r](\overrightarrow{x})\sigma.P,\gamma_r)$ can only perform a broadcast input on action α and that it always refuses input on any other action type $\beta \neq \alpha$.

The behaviour of *unicast output* and *unicast input* is defined by the first five rules of Table 2. These rules are similar to the ones already presented for broadcast output and broadcast input.

The other rules of Table 2 describe the behaviour of other process operators, namely *choice* $P+Q$, *parallel composition* $P|Q$, *guard* and *recursion*. The term $P+Q$ identifies

[6] We let $[\![\cdot]\!]_\gamma$ denote the evaluation function of an expression/predicate with respect to the store γ.

$$\frac{[\![\pi_s]\!]_\gamma = \pi_s' \quad [\![\overrightarrow{e}]\!]^t_\gamma = \overrightarrow{v} \quad \mathbf{p} = \sigma(\gamma)}{\mathbb{C}[(\alpha[\pi_s]\langle\overrightarrow{e}\rangle\sigma.P,\gamma),\alpha[\pi_s']\langle\overrightarrow{v}\rangle,\gamma] = (P,\mathbf{p})}\ \textbf{Out}$$

$$\frac{[\![\pi_s]\!]_\gamma = \pi_s' \quad [\![\overrightarrow{e}]\!]^t_\gamma = \overrightarrow{v} \quad \ell \neq \alpha[\pi_s']\langle\overrightarrow{v}\rangle,\gamma}{\mathbb{C}[(\alpha[\pi_s]\langle\overrightarrow{e}\rangle\sigma.P,\gamma),\ell] = \emptyset}\ \textbf{Out-F}$$

$$\frac{\gamma_r \models \pi_s \quad \gamma_s \models \pi_r[\overrightarrow{v}/\overrightarrow{x}] \quad \mathbf{p} = \sigma[\overrightarrow{v}/\overrightarrow{x}](\gamma_2)}{\mathbb{C}[(\alpha[\pi_r](\overrightarrow{x})\sigma.P,\gamma_r),\alpha[\pi_s](\overrightarrow{v}),\gamma_s] = (P[\overrightarrow{v}/\overrightarrow{x}],\mathbf{p})}\ \textbf{In}$$

$$\frac{\gamma_r \not\models \pi_s \vee \gamma_s \not\models \pi_r[\overrightarrow{v}/\overrightarrow{x}]}{\mathbb{C}[(\alpha[\pi_r](\overrightarrow{x})\sigma.P,\gamma_r),\alpha[\pi_r](\overrightarrow{v}),\gamma_r] = \emptyset}\ \textbf{In-F1} \qquad \frac{\ell \neq \alpha[\pi_s](\overrightarrow{v}),\gamma_s}{\mathbb{C}[(\alpha[\pi_r](\overrightarrow{x})\sigma.P,\gamma_r),\ell] = \emptyset}\ \textbf{In-F2}$$

$$\frac{\mathbb{C}[(P,\gamma),\ell] = \mathscr{C}_1 \quad \mathbb{C}[(Q,\gamma),\ell] = \mathscr{C}_2}{\mathbb{C}[(P+Q,\gamma),\ell] = \mathscr{C}_1 \oplus \mathscr{C}_2}\ \textbf{Plus}$$

$$\frac{\gamma \models \pi \quad \mathbb{C}[(P,\gamma),\ell] = \mathscr{C}}{\mathbb{C}[([\pi]P,\gamma),\ell] = \mathscr{C}}\ \textbf{Guard} \qquad \frac{\gamma \not\models \pi}{\mathbb{C}[([\pi]P,\gamma),\ell] = \emptyset}\ \textbf{Guard-F}$$

$$\frac{\mathbb{C}[(P,\gamma),\ell] = \mathscr{C}_1 \quad \mathbb{C}[(Q,\gamma),\ell] = \mathscr{C}_2}{\mathbb{C}[(P|Q,\gamma),\ell] = \mathscr{C}_1|Q \oplus P|\mathscr{C}_2}\ \textbf{Par} \qquad \frac{A \stackrel{\triangle}{=} P \quad \mathbb{C}[(P,\gamma),\ell] = \mathscr{C}}{\mathbb{C}[(A,\gamma),\ell] = \mathscr{C}}\ \textbf{Rec}$$

Table 2: Operational semantics of components (Part 2)

a process that can behave either as P or as Q. The rule **Plus** states that the components that are reachable by $(P+Q,\gamma)$ are the ones that can be reached either by (P,γ) or by (Q,γ). In this rule we use $\mathscr{C}_1 \oplus \mathscr{C}_2$ to denote the function that maps each term C to $\mathscr{C}_1(C)+\mathscr{C}_2(C)$, for any $\mathscr{C}_1,\mathscr{C}_2 \in [\textsc{Comp} \to \mathbb{R}_{\geq 0}]$.

In $P|Q$ the two composed processes interleave for all the transition labels. In the rule the following notations are used:

- for each component C and process Q we let:

$$C|Q = \begin{cases} \mathbf{0} & C \equiv \mathbf{0} \\ (P|Q,\gamma) & C \equiv (P,\gamma) \end{cases}$$

  $Q|C$ is symmetrically defined.
- for each $\mathscr{C} : \textsc{Comp} \to \mathbb{R}_{\geq 0}$ and process Q, $\mathscr{C}|Q$ (resp. $Q|\mathscr{C}$) denotes the function that maps each term of the form $C|Q$ (resp. $Q|C$) to $\mathscr{C}(C)$, while the others are mapped to 0;

Rule **Rec** is standard. The behaviour of $([\pi]P,\gamma)$ is regulated by rules **Guard** and **Guard-F**. The first rule states that $([\pi]P,\gamma)$ behaves exactly like (P,γ) when γ satisfies predicate π. However, in the first case the *guard* is removed when a transition is performed. In contrast, no component is reachable when the *guard* is not satisfied (rule **Guard-F**).

The following lemma guarantees that for any C and for any ℓ $\mathbb{C}[C,\ell]$ is either a probability distribution or the 0 constant function $\emptyset$.

3.2 Operational semantics of collectives

The operational semantics of a *collective* is defined via the function

$$\mathbb{N}_\varepsilon : \textsc{Col} \times \textsc{Lab}_I \to [\textsc{Col} \to \mathbb{R}_{\geq 0}]$$

that is formally defined in Table 3, where we use a straightforward adaptation of the notations introduced in the previous section. This function shows how a collective reacts when a broadcast/unicast message is received. Indeed, $\textsc{Lab}_I$ denotes the subset of $\textsc{Lab}$ with only input labels:

$$\begin{array}{rll} \ell ::= & \alpha^\star[\pi_s](\overrightarrow{v}),\gamma & \text{Broadcast Input} \\ \mid & \alpha[\pi_s](\overrightarrow{v}),\gamma & \text{Unicast Input} \end{array}$$

Given a collective N and an input label $\ell \in \textsc{Lab}_I$, function $\mathbb{N}_\varepsilon[N,\ell]$ returns a function $\mathscr{N}$ that associates each collective N' reachable from N via ℓ with a value in $\mathbb{R}_{\geq 0}$. If ℓ is a broadcast input $(\alpha^\star[\pi_s](\overrightarrow{v}),\gamma)$ this value represents the probability that the collective is reachable after ℓ. When ℓ is a unicast input $\alpha[\pi_s](\overrightarrow{v}),\gamma$, $\mathscr{N}(N')$ is the weight that will be used, at the level of systems, to compute the probability that N' is selected after ℓ. Note that this difference is due from the fact that while the probability to receive a *broadcast input* can be computed *locally* (each component identifies its own probability), to compute the probability to receive a *unicast input* the complete collective

$$\frac{}{\mathbb{N}_\varepsilon[\mathbf{0},\ell]=\emptyset}\ \textbf{Zero}$$

$$\frac{\mathbb{C}[(P,\gamma),\alpha^\star[\pi_s](\overrightarrow{\nu}),\gamma]=\mathscr{N}\quad \mathscr{N}\neq\emptyset\quad \varepsilon=\langle\mu_p,\mu_w,\mu_r,\mu_u\rangle}{\mathbb{N}_\varepsilon[(P,\gamma),\alpha^\star[\pi_s](\overrightarrow{\nu}),\gamma]=\frac{\mu_p(\gamma,\alpha^\star)}{\oplus\mathscr{N}}\cdot\mathscr{N}+[(P,\gamma)\mapsto(1-\mu_p(\gamma,\alpha^\star)]}\ \textbf{Comp-B-In}$$

$$\frac{\mathbb{C}[(P,\gamma),\alpha^\star[\pi_s](\overrightarrow{\nu}),\gamma]=\emptyset}{\mathbb{N}_\varepsilon[(P,\gamma),\alpha^\star[\pi_s](\overrightarrow{\nu}),\gamma]=[(P,\gamma)\mapsto 1]}\ \textbf{Comp-B-In-F}$$

$$\frac{\mathbb{C}[(P,\gamma_2),\alpha[\pi_s](\overrightarrow{\nu}),\gamma_1]=\mathscr{N}\quad \mathscr{N}\neq\emptyset\quad \varepsilon=\langle\mu_p,\mu_w,\mu_r,\mu_u\rangle}{\mathbb{N}_\varepsilon[(P,\gamma_2),\alpha[\pi_s](\overrightarrow{\nu}),\gamma_1]=\mu_w(\gamma_1,\gamma_2,\alpha)\cdot\frac{\mathscr{N}}{\oplus\mathscr{N}}}\ \textbf{Comp-In}$$

$$\frac{\mathbb{C}[(P,\gamma_2),\alpha[\pi_s](\overrightarrow{\nu}),\gamma_1]=\emptyset}{\mathbb{N}_\varepsilon[(P,\gamma_2),\alpha[\pi_s](\overrightarrow{\nu}),\gamma_1]=\emptyset}\ \textbf{Comp-In-F}$$

$$\frac{\mathbb{N}_\varepsilon[N_1,\alpha^\star[\pi_s](\overrightarrow{\nu}),\gamma]=\mathscr{N}_1\quad \mathbb{N}_\varepsilon[N_2,\alpha^\star[\pi_s](\overrightarrow{\nu}),\gamma]=\mathscr{N}_2}{\mathbb{N}_\varepsilon[N_1\parallel N_2,\alpha^\star[\pi_s](\overrightarrow{\nu}),\gamma]=\mathscr{N}_1\parallel\mathscr{N}_2}\ \textbf{B-In-Sync}$$

$$\frac{\mathbb{N}_\varepsilon[N_1,\alpha[\pi_s](\overrightarrow{\nu}),\gamma]=\mathscr{N}_1\quad \mathbb{N}_\varepsilon[N_2,\alpha[\pi_s](\overrightarrow{\nu}),\gamma]=\mathscr{N}_2}{\mathbb{N}_\varepsilon[N_1\parallel N_2,\alpha[\pi_s](\overrightarrow{\nu}),\gamma]=\mathscr{N}_1\parallel N_2\oplus N_1\parallel\mathscr{N}_2}\ \textbf{In-Sync}$$

Table 3: Operational semantics of collective

is needed. Function $\mathbb{N}_\varepsilon$ is also parametrised with respect to the *evaluation function* ε, obtained from the environment where the collective operates, that is used to compute the above mentioned *weights*.

The first four rules in Table 3 describe the behaviour of the single component at the level of collective. Rule **Zero** is similar to rule **Nil** of Table 1 and states that inactive component $\mathbf{0}$ cannot perform any action. Rule **Comp-B-In** states that if (P,γ) can receive a message sent via a broadcast with activity α ($\mathbb{C}[(P,\gamma),\alpha^\star[\pi_s](\overrightarrow{v}),\gamma] = \mathscr{N} \neq \emptyset$) then the component receives the message with probability $\mu_p(\gamma,\alpha^\star)$ while the message is not received with probability $1-\mu_p(\gamma,\alpha^\star)$. In the first case, the resulting function is renormalised by $\oplus\mathscr{N}$ to indicate that each element in P receives the message with the same probability. There we use $\oplus\mathscr{N}$ to denote $\sum_{N\in \text{COL}}\mathscr{N}(N)$. On the contrary, rule **Comp-B-In-F** states that if (P,γ) is not able to receive a broadcast message, ($\mathbb{C}[(P,\gamma),\alpha^\star[\pi_s](\overrightarrow{v}),\gamma] = \emptyset$), with probability 1 the message is received while the component remains unchanged.

Rule **Comp-In** is similar to **Comp-B-In**. It simply lifts the transition at the level of component to the level of collective while the resulting function is multiplied by the weight $\mu_p(\gamma_1,\gamma_2,\alpha)$. The latter is the probability that this component is selected for the synchronisation. As in **Comp-B-In**, function $\mathscr{N}$ is divided by $\oplus\mathscr{N}$ to indicate that any possible receiver in P is selected with the same probability. Rule **Comp-In-F** is applied when a component is not involved in a synchronisation.

Rule **B-In-Sync** states that that two collectives N_1 and N_2 that operate in parallel synchronise while performing a broadcast input. This models the fact that the input can be potentially received by both of the collectives. In this rule we let $\mathscr{N}_1 \parallel \mathscr{N}_2$ denote the function associating the value $\mathscr{N}_1(N_1)\cdot\mathscr{N}_2(N_2)$ with each term of the form $N_1 \parallel N_2$ and 0 with all the other terms. We can observe that if

$$\mathbb{N}_\varepsilon[N,\alpha^\star[\pi_s](\overrightarrow{v}),\gamma] = \mathscr{N}$$

then, as we have already observed for rule **Comp-B-In**, $\oplus\mathscr{N} = 1$ and $\mathscr{N}$ is in fact a probability distribution over COL.

Rule **In-Sync** controls the behaviour associated with unicast input and it states that a collective of the form $N_1 \parallel N_2$ performs a *unicast input* if this is performed either in N_1 or in N_2. This is rendered in the semantics as an interleaving rule, where for each $\mathscr{N} : \text{COL} \to \mathbb{R}_{\geq 0}$, $\mathscr{N} \parallel N_2$ denotes the function associating $\mathscr{N}(N_1)$ with each collective of the form $N_1 \parallel N_2$ and 0 with all other collectives.

3.3 Operational semantics of systems

The operational semantics of systems is defined via the function

$$\mathbb{S}_t : \text{SYS} \times \text{LAB}_S \to [\text{SYS} \to \mathbb{R}_{\geq 0}]$$

that is formally defined in Table 4. This function only considers synchronisation labels LAB_S:

$$\begin{array}{rll} \ell ::= & \alpha^\star[\pi_s]\langle\overrightarrow{v}\rangle,\gamma & \text{Broadcast Output} \\ \mid & \tau[\alpha[\pi_s]\langle\overrightarrow{v}\rangle,\gamma] & \text{Unicast Synchronization} \end{array}$$

The behaviour of a CARMA system is defined in terms of a *time-inhomogeneous Action Labelled Markov Chain* whose transition matrix is defined by function $\mathbb{S}_t$. For any system S and for any label $\ell \in \text{LAB}_S$, if $\mathbb{S}_t[S,\ell] = \mathscr{S}$ then $\mathscr{S}(S')$ is the rate of the transition from S to S'. When $\mathscr{S}(S') = 0$ then S' is not reachable from S via ℓ.

The first rule is **Sys-B**. This rule states that, when $\varepsilon = \langle \mu_p, \mu_w, \mu_r, \mu_u \rangle = \rho(t, \gamma_g, N)$, a system of the form N **in** (γ_g, ρ) at time t can perform a *broadcast output* when there is a component $C \in N$ that performs the output while the remaining part of the collective $(N - C)$ performs the complementary input. The outcome of this synchronisation is computed by the function $\mathsf{bSync}_\varepsilon$ defined below:

$$\frac{\varepsilon = \langle \mu_p, \mu_w, \mu_r, \mu_u \rangle \quad \mathbb{C}[C, \alpha^\star[\pi_s]\langle \overrightarrow{v} \rangle, \gamma] = \mathscr{C} \quad \mathbb{N}_\varepsilon[N, \alpha^\star[\pi_s](\overrightarrow{v}), \gamma] = \mathscr{N}}{\mathsf{bSync}_\varepsilon(C, N, \alpha^\star[\pi_s]\langle \overrightarrow{v} \rangle, \gamma) = \mu_r(\gamma, \alpha^\star[\pi_s]\langle \overrightarrow{v} \rangle, \gamma) \cdot \mathscr{C} \parallel \mathscr{N}}$$

This function combines the outcome of the broadcast output performed by C, ($\mathscr{C}$) with the complementary input performed by N ($\mathscr{N}$), the result is then multiplied by the rate of the action induced by the environment $\mu_r(\gamma_C, \alpha^\star[\pi_s]\langle \overrightarrow{v} \rangle, \gamma)$. Note that, since both $\mathscr{C}$ and $\mathscr{N}$ are probability distributions, the same is true for $\mathscr{C} \parallel \mathscr{N}$.

To compute the total rate of a synchronisation we have to sum the outcome above for all the possible senders $C \in N$ multiplied by the multiplicity of C component in N ($N(C)$). After the synchronisation, the global store is updated and a new collective can be created according to function μ_u. In rule **Sys-B** the following notations are used. For each collective N_2, $\mathscr{N} : \text{COL} \to \mathbb{R}_{\geq 0}$, $\mathscr{S} : \text{SYS} \to \mathbb{R}_{\geq 0}$ and $\mathbf{p} \in \mathit{Dist}(\Gamma)$ we let $\mathscr{N}$ **in** $(\mathbf{p}, \rho)$ denote the function mapping each system N **in** (γ, ρ) to $\mathscr{N}(N) \cdot \mathbf{p}(\gamma)$.

The second rule is **Sys** that regulates unicast synchronisations, which is similar to **Sys-B**. However, there function $\mathsf{uSync}_\varepsilon$ is used. This function is defined below:

$$\frac{\varepsilon = \langle \mu_p, \mu_w, \mu_r, \mu_u \rangle \quad \mathbb{C}[C, \alpha^\star[\pi_s]\langle \overrightarrow{v} \rangle, \gamma] = \mathscr{C} \quad \mathbb{N}_\varepsilon[N, \alpha^\star[\pi_s](\overrightarrow{v}), \gamma] = \mathscr{N} \neq \emptyset}{\mathsf{uSync}_\varepsilon(C, N, \alpha^\star[\pi_s]\langle \overrightarrow{v} \rangle, \gamma) = \mu_r(\gamma_C, \alpha^\star[\pi_s]\langle \overrightarrow{v} \rangle, \gamma) \cdot \mathscr{C} \parallel \frac{\mathscr{N}}{\oplus \mathscr{N}}}$$

$$\frac{\mathbb{N}_\varepsilon[N, \alpha^\star[\pi_s](\overrightarrow{v}), \gamma] = \emptyset}{\mathsf{uSync}_\varepsilon(C, N, \alpha^\star[\pi_s]\langle \overrightarrow{v} \rangle, \gamma) = \emptyset}$$

Similarly to $\mathsf{bSync}_\varepsilon$, this function combines the outcome of a unicast output performed by C, ($\mathscr{C}$) with the complementary input performed by N ($\mathscr{N}$). The result is then multiplied by the rate of the action induced by the environment $\mu_r(\gamma_C, \alpha^\star[\pi_s]\langle \overrightarrow{v} \rangle, \gamma)$. However, in $\mathsf{uSync}_\varepsilon$ we have to renormalise $\mathscr{N}$ by the value $\oplus \mathscr{N}$. This guarantees that the total synchronisation rate does not exceeds the capacity of the sender. Note that, $\mathscr{N}$ is not a probability distribution while $\frac{\mathscr{N}}{\oplus \mathscr{N}}$ is.

4 CARMA implementation

To support simulation of CARMA models, a prototype simulator has been developed. This simulator, which has been implemented in Java, can be used to perform stochastic simulation and will be the basis for the implementation of other analysis techniques. An

$$\frac{\rho(t,\gamma_g,N)=\varepsilon=\langle\mu_p,\mu_w,\mu_r,\mu_u\rangle \quad \mu_u(\gamma_g,\alpha^\star)=(\sigma,N') \quad \sum_{C\in N} N(C)\cdot \mathsf{bSync}(C,N-C,\alpha^\star[\pi_s]\langle\overrightarrow{v}\rangle,\gamma)=\mathscr{N}}{\mathbb{S}_t[N \textbf{ in } (\gamma_g,\rho),\alpha^\star[\pi_s]\langle\overrightarrow{v}\rangle,\gamma]=\mathscr{N} \parallel N' \textbf{ in } (\sigma(\gamma_g),\rho)} \textbf{ Sys-B}$$

$$\frac{\rho(t,\gamma_g,N)=\varepsilon=\langle\mu_p,\mu_w,\mu_r,\mu_u\rangle \quad \mu_u(\gamma_g,\alpha^\star)=(\sigma,N') \quad \sum_{C\in N} N(C)\cdot \mathsf{uSync}(C,N-C,\tau[\alpha[\pi_s]\langle\overrightarrow{v}\rangle,\gamma])=\mathscr{N}}{\mathbb{S}_t[N \textbf{ in } (\gamma_g,\rho),\tau[\alpha[\pi_s]\langle\overrightarrow{v}\rangle,\gamma]]=\mathscr{N} \parallel N' \textbf{ in } (\sigma(\gamma_g),\rho)} \textbf{ Sys}$$

Table 4: Operational Semantics of Systems.

Eclipse plug-in for supporting specification and analysis of CAS in CARMA has also been developed. In this plug-in, CARMA systems are specified by using an appropriate high-level language for designers of CAS, named the CARMA *Specification Language*. This is mapped to the process algebra, and hence will enable qualitative and quantitive analysis of CAS during system development by enabling a design workflow and analysis pathway. The intention of this high-level language is not to add to the expressiveness of CARMA, which we believe to be well-suited to capturing the behaviour of CAS, but rather to ease the task of modelling for users who are unfamiliar with process algebra and similar formal notations. Both the simulator and the Eclipse plug-in are available at https://quanticol.sourceforge.net/.

In the rest of this section, we first describe the CARMA *Specification Language* then an overview of the CARMA Eclipse Plug-in is provided. In Section 5 we will show how the *Bike Sharing System* considered in Section 2 can be modelled, simulated and analysed with the CARMA tools.

4.1 CARMA specification l anguage

In this section we present the language that supports the design of CAS in CARMA. To describe the main features of this language, following the same approach used in Section 2, we will use the *Bike Sharing System*.

Each CARMA specification, also called a CARMA *model*, provides definitions for:

- structured *data types* and the relative *functions*;
- prototypes of *components*;
- *systems* composed of collective and environment;
- *measures*, that identify the relevant data to *measure* during simulation runs.

Data types. Three basic types are natively supported in our specification language. These are: `bool`, for booleans, `int`, for integers, and `real`, for real values. However, to model complex structures, it is often useful to introduce custom types. In a CARMA specification two kind of custom types can be declared: *enumerations* and *records*.

Like in many other programming languages, an *enumeration* is a data type consisting of a set of *named values*. The enumerator names are identifiers that behave as

constants in the language. An attribute (or variable) that has been declared as having an enumerated type can be assigned any of the enumerators as its value. In other words, an enumerated type has values that are different from each other, and that can be compared and assigned, but which are not specified by the programmer as having any particular concrete representation. The syntax to declare a new *enumeration* is:

enum *name* = $elem_1, \ldots, elem_n$;

where *name* is the name of the declared enumeration while $elem_i$ are its value names. Enumeration names start with a capitalised letter while the enumeration values are composed by only capitalised letters.

Example 8. Enumerations can be used to define predefined set of values that can be used in the specification. For instance one can introduce an enumeration to identify the possible four directions of movement:

```
enum Direction = NORTH, SOUTH, EAST, WEST;
```

To declare aggregated data structures, a CAS designer can use *records*. A record consists of a sequence of a set of typed fields:

record *name* = [$type_1$ $field_1, \ldots,$ $type_n$ $field_n$];

Each field has a type $type_i$ and a name $field_i$: $type_i$ can be either a built-in type or one of the new declared types in the specification; $field_i$ can be any valid identifier.

Example 9. Record can be used to model structured elements. For instance, a position over a grid can be rendered as follows:

```
record Position = [ int x, int y];
```

A record can be created by assigning a value to each field, within square brackets:

[$field_1 := expression_1, \ldots,$ $field_n := expression_n$]

Example 10. The instantiation of a location referring to the point located at $(0,0)$ has the following form:

```
[ x:=0 , y:=0 ]
```

Given a variable (or attribute) having a record type, each field can be accessed using the *dot* notation:

$variable.field_i$

Constants and Functions. A CARMA specification can also contain *constants* and *functions* declarations having the following syntax:

```
const name = expression;

fun type name( type_1 arg_1 ,..., type_k arg_k ) {
  ...
}
```

where the body of an expression consists of standard statements in a high-level programming language. The type of a constant is not declared but inferred directly from the assigned expression.

Example 11. A constant can be used to represent the number of *zones* in the *Bike Sharing System*:

```
const ZONES = 5;
```

Moreover, functions can be used to perform complex computations that cannot be done in a single expression:

```
fun real ReceivingProb( int size ) {
  if (size != 0) {
    return 1.0/real(size);
  } else {
    return 0.0;
  }
}
```

Components prototype. A *component prototype* provides the general structure of a component that can be later instantiated in a CARMA system. Each prototype is parameterised with a set of typed parameters and defines: the store; the component's behaviour and the initial configuration. The syntax of a *component prototype* is:

```
component name( type_1 arg_1 ,..., type_n arg_n ) {
    store { ···
        attr_kind aname_i := expression_i; ···
    }
    behaviour { ···
        proc_i = pdef_i; ···
    }
    init { P_1 | ··· | P_w }
}
```

Each component prototype has a possibly empty list of arguments. Each argument arg_i has a type $type_i$ that can be one of the built-in types (`bool`, `int` and `real`), a custom type (an enumeration or record), or the type `process` that indicates a component behaviour. These arguments can be used in the body of the component. The latter consists of three (optional) blocks: `store`, `behaviour` and `init`.

The block `store` defines the list of attributes (and their initial values) exposed by a component. Each attribute definition consists of an attribute kind *attr_kind* (that can

be either `attrib` or `const`), a *name* and an expression identifying the initial attribute value. When an attribute is declared as `const`, it cannot be changed. The actual type of an attribute is not declared but inferred from the expression providing its initialisation value.

The block `behaviour` is used to define the processes that are specific to the considered components and consists of a sequence of definitions of the form

$$proc_i \ = \ pdef_i ;$$

where $proc_i$ is the process name while $pdef_i$ is its definition having the following syntax[7]:

$$\begin{aligned}
pdef \ ::= \ & pdef + pdef \\
\mid \ & \texttt{[}\ expr\ \texttt{]}\ pdef \\
\mid \ & act\,.\,proc
\end{aligned}$$

$$\begin{aligned}
act \ ::= \ & act_name\texttt{[}\ expr\ \texttt{]<}expr_1,\ldots,\ expr_n\texttt{>\{}aname_1 := expr'_1,\ldots,aname_k := expr'_k\texttt{\}} \\
\mid \ & act_name\texttt{*[}\ expr\ \texttt{]<}expr_1,\ldots,\ expr_n\texttt{>\{}aname_1 := expr'_1,\ldots,aname_k := expr'_k\texttt{\}} \\
\mid \ & act_name\texttt{[}\ expr\ \texttt{]}(var_1,\ldots,\ var_n)\texttt{\{}aname_1 := expr'_1,\ldots,aname_k := expr'_k\texttt{\}} \\
\mid \ & act_name\texttt{*[}\ expr\ \texttt{]}(var_1,\ldots,\ var_n)\texttt{\{}aname_1 := expr'_1,\ldots,aname_k := expr'_k\texttt{\}}
\end{aligned}$$

Finally, block `init` is used to specify the initial behaviour of a component. It consists of a sequence of terms P_i separated by the symbol `|`. Each P_i can be a process defined in the block `behaviour`, `kill` or `nil`.

Example 12. The prototypes for Station, Users and Arrival components, already described in Example 2, can be defined as follows:

```
component Station( int loc , int capacity , int available )
{
  store {
    attrib loc := loc;
    attrib available := available;
    attrib capacity := capacity;
  }
  behaviour {
    G = [my.available >0]
                get<>{ my.available := my.available -1 }.G;
    R = [my.available <my.capacity]
                ret<>{ my.available := my.available+1 }.R;
  }
  init {
    G|R
  }
}
```

[7] All the operators are right associative and presented in the order of priority.

```
component User( int loc , int dest ) {
  store {
    attrib loc := loc;
    attrib dest := dest;
  }
  behaviour {
    P = get[ my.loc == loc ]().B;
    B = move*[ false ]<>{ my.loc := my.dest }.W;
    W = ret[ my.loc == loc ]().kill;
  }
  init {
    P
  }
}

component Arrival( int loc ) {
  store {
    attrib loc := loc;
  }
  behaviour {
    A = arrival*[false]<>.A;
  }
  init {
    A
  }
}
```

System definitions. A system definition consists of two blocks, namely `collective` and `environment`, that are used to declare the collective in the system and its environment, respectively:

```
system name {
  collective {
    inist_stmt
  }
  environment { ···
  }
}
```

Above, *inist_stmt* indicates a sequence of commands that are used to instantiate components. The basic command to create a new component is:

new *name*($expr_1$,...,$expr_n$)

where *name* is the name of a component prototype. However, in a system a large number of collectives can occur. For this reason, our specification language provides specific constructs for the instantiation of multiple copies of a component. A first construct is the *range operator*. This operator is of the form:

[$expr_1$: $expr_2$: $expr_3$]

and can be used as an argument of type integer. It is equivalent to a sequence of integer values starting from $expr_1$, ending at $expr_2$. The element $expr_3$ (that is optional) indicates the step between two elements in the sequence. When $expr_3$ is omitted, value 1 is assumed. The *range operator* can be used where an integer parameter is expected. This is equivalent to having multiple copies of the same instantiation command where each element in the sequence replaces the command.

For instance, assuming `ZONES` to be the constant identifying the number of zones in the city, while `CAPACITY` and `INITIAL_AVAILABILITY` refer to the station capacity and to the initial availability, respectively, the instantiation of the stations can be modelled as:

```
new Station( [0:ZONES-1] , CAPACITY, INITIAL_AVAILABILITY );
```

The command above is equivalent to:

```
new Station( 0 , CAPACITY, INITIAL_AVAILABILITY );
        ⋮
new Station( ZONES-1 , CAPACITY, INITIAL_AVAILABILITY );
```

Two other commands are used to control instantiation of components. These are:

```
for ( var_name = expr1 ; expr2 ; expr3 ) {
    inist_stmt
}

if ( expr ) {
    inist_stmt
} else {
    inist_stmt
}
```

The former is used to iterate an instantiation block for a given number of times while the latter can be used to differentiate the instantiation depending on a given condition.

Example 13. The following block can be used to instantiate `SITES` copies of component `Station` at each zone. The same block instantiates a component `Arrival` at each zone:

```
collective {
  for ( i ; i<ZONES ; 1 ) {
    for ( j ; j<SITES ; 1 ) {
      new Station( i , CAPACITY, INITIAL_AVAILABILITY );
    }
    new Arrival(i);
  }
}
```

The syntax of a block `environment` is the following:

```
environment {
  store { ··· }
  prob { ··· }
```

```
  weight { ··· }
  rate { ··· }
  update { ··· }
}
```

The block `store` defines the *global store* and has the same syntax as the similar block already considered in the component prototypes.

Example 14. In the *Bike Sharing System* we use a global attribute to count the amount of *active users* in the system:

```
store {
  attrib users := 0;
}
```

Blocks `prob` and `weight` are used to compute the probability to receive a message. Syntax of `prob` is the following:

```
prob { ···
    [guard_i] act_i : expr_i; ···
    default : expr;
}

weight { ···
    [guard_i] act_i : expr_i; ···
    default : expr;
}
```

In the above, each $guard_i$ is a boolean expression over the global store and the stores of the two interacting components, i.e. the sender and the receiver, while act_i denotes the action used to interact. In $guard_i$ attributes of sender and receiver are referred to using `sender.a` and `receiver.a`, while the values published in the global store are referenced by using `global.a`. This probability value may depend on the number of components in a given state. To compute this value, expressions of the following form can be used:

```
#{ Π | expr }
```

This expression denotes the number of components in the system satisfying boolean expression *expr* where a process of the form Π is executed. In turn, Π is a pattern of the following form:

$$\Pi ::= * \mid *[\ proc\ ] \mid comp[\ *\ ] \mid comp[\ proc\ ]$$

Example 15. In our example the block `weight` can be instantiated as follows:

```
weight{
  [receiver.loc==sender.loc] get: 1;
  [receiver.loc==sender.loc] ret: 1;
  default: 0;
}
```

Above, we say that each user in a zone receives a bike/parking slot with the same probability. All the other interactions are disabled having the associated weight equal to 0.

Block `rate` is similar and it is used to compute the rate of an unicast/broadcast output. This represents a function taking as parameter the local store of the component performing the action and the action type used. Note that the environment can disable the execution of a given action. This happens when evaluation of block `rate` (resp. `prob`) is 0. Syntax of `rate` is the following:

```
rate { ···
    [guard_i] act_i : expr_i; ···
    default : expr;
}
```

Differently from `prob`, in `rate` guards $guard_i$ are evaluated by considering only the attributes defined in the store of the component performing the action, referenced as `sender.a`, or in the global store, accessed via `global.a`.

Example 16. In our example `rate` can be defined as follow:

```
rate{
  [true] get: get_rate;
  [true] ret: ret_rate;
  [true] move*: move_rate;
  [true] arrival*:
    (global.users<TOTAL_USERS?arrival_rate:0.0);
  [true] default: 1;
}
```

Above we say that actions `move*`, `get` and `ret` are executed at a constant rate. Rate of user arrivals depends on the number of active users. Action `arrival*` is executed with rate `arrival_rate` when the total number of users active in the system is less than `TOTAL_USERS`. Otherwise, the same action is disabled (i.e. executed with rate 0.0).

Finally, the block `update` is used to update the global store and to install a new collective in the system. Syntax of `update` is:

```
update { ···
    [guard_i] act_i : attr_updt_i; inst_cmd_i; ···
}
```

As for `rate`, guards in the `update` block are evaluated on the store of the component performing the action and on the global store. However, the result is a sequence of attribute assignments followed by an instantiation command (above considered in the collective instatiation). If none of the guards are satisfied, or the performed action is not listed, the global store is not changed and no new collective is instantiated. In both cases, the collective generating the transition remains in operation. This function is particularly useful for modelling the arrival of new agents into a system.

Example 17. In our scenario block `update` is used to model the arrival of new users and the exit of existing ones. It is defined as follows:

```
update {
  [true] arrival*: users := global.users+1 , new User(
      sender.loc , U[0:ZONES-1] );
  [true] ret: users := global.users-1;
}
```

When action `arrival*` is performed a component associated with a new user is created in the same location as the sender (see Example 3). The destination of the new user is probabilistically selected. Indeed, above we use `U[0:ZONES-1]` to indicate the uniform probability over the integer values between `0` and `ZONES-1` (included). When a bike is returned, the user exits from the system (process `kill` is enabled) and the global attribute `users` is updated accordingly.

Measure definitions. To extract observations from a model, a CARMA specification also contains a set of *measures*. Each measure is defined as:

measure *m_name*[var_1=$range_1$, ... , var_n=$range_n$] = *expr*;

Expression *expr* can be used to count, by using expressions of the form `#{` *Π* `|` *expr* `}` already described above, or to compute statistics about attribute values of components operating in the system: `min{` *expr* `|` *guard* `}`, `max{` *expr* `|` *guard* `}` and `avg{` *expr* `|` *guard* `}`. These expressions are used to compute the minimum/maximum/average value of expression *expr* evaluated in the store of all the components satisfying boolean expression *guard*, respectively.

Example 18. In our scenario, we are interested in measuring the number of available bikes in a zone. For this reason, the following measures are used:

```
measure AverageBikes[l:=0:4] =
    avg{ my.available | my.loc == l };
measure MinBikes[l:=0:4] =
    min{ my.available | my.loc == l };
measure MaxBikes[l:=0:4] =
    max{ my.available | my.loc == l };
```

4.2 CARMA Eclipse Plug-in

The CARMA specification language is implemented as an Eclipse plug-in using the Xtext framework. It can be downloaded using the standard procedure in Eclipse by pointing to the update site at http://quanticol.sourceforge.net/updates/[8]. After the installation, the CARMA editor will open any file in the workspace with the `carma` extension.

Given a CARMA specification, the CARMA Eclipse Plug-in automatically generates the Java classes providing the machinery to simulate the model. This generation procedure can be specialised to enable the use of different kind of simulators. Currently, a simple ad-hoc simulator, is used. The simulator provides generic classes for representing simulated systems (named here *models*). To perform the simulation each *model*

[8] Detailed installation instructions can be found at http://quanticol.sourceforge.net

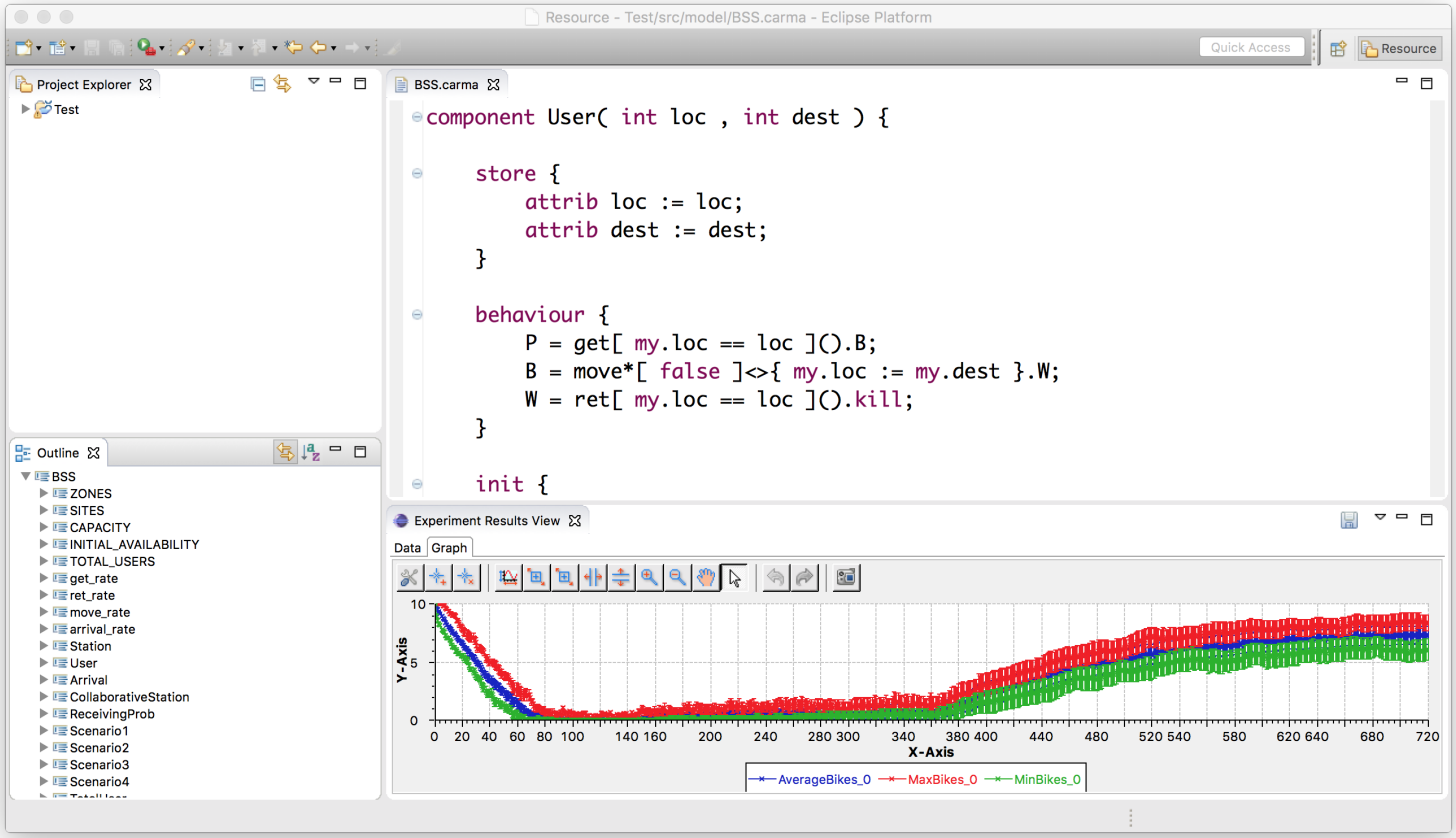

Fig. 1: A screenshot of the CARMA Eclipse plug-in.

provides a collection of *activities* each of which has its own *execution rate*. The simulation environment applies a standard *kinetic Monte-Carlo* algorithm to select the next activity to be executed and to compute the execution time. The execution of an *activity* triggers an update in the simulation model and the simulation process continues until a given simulation time is reached. In the classes generated from a CARMA specification, these activities correspond to the *actions* that can be executed by processes located in the system components. Each of these activities in fact mimics the execution of a transition of the CARMA operational semantics. Specific *measure functions* can be passed to the simulation environment to collect simulation data at given intervals. To perform statistical analysis of collected data the *Statistics package* of *Apache Commons Math Library* is used[9].

To access the simulation features, a user can select the menu *Carma→Simulation*. When this menu is selected, a dialogue box pops up to choose the simulation parameters (see Figure 2). This dialogue box is automatically populated with appropriate values from the model. When the selection of the simulation parameters is completed, the simulation is started. The results are reported within the *Experiment Results View* (see Figure 3). Two possible representation are available. The former, on the left side of Figure 3, provides a graphical representation of collected data; the latter, on the right side of Figure 3, shows average and standard deviation of the collected values, which correspond to the *measures* selected during the simulation set-up, are reported in a tabular form. These values can then be exported in CSV format and used to build suitable plots in the preferred application.

5 CARMA tools in action

In this section we present the *Bike Sharing System* in its entirety and demonstrate the quantitative analysis which can be undertaken on a CARMA model. One of the main advantages of the fact that we structure a CARMA system specification in two parts – a collective and an environment – is that we can evaluate the same collective in different enclosing environments.

We now consider a scenario with 5 zones and instantiate the environment of the *Bike Sharing Systems* with respect to two different specifications for the environment:

Scenario 1: Users always arrive in the system at the same rate;
Scenario 2: User arrival rate is higher at the beginning (modelling the fact that bikes are mainly used in the morning) and then decreases.

The first scenario is the one presented in Section 4 and reported below for completeness:

```
system Scenario1 {
  collective {
    for(i;i<ZONES;1) {
      for(j;j<SITES;1) {
        new Station(i,CAPACITY,INITIAL_AVAILABILITY);
```

[9] http://commons.apache.org

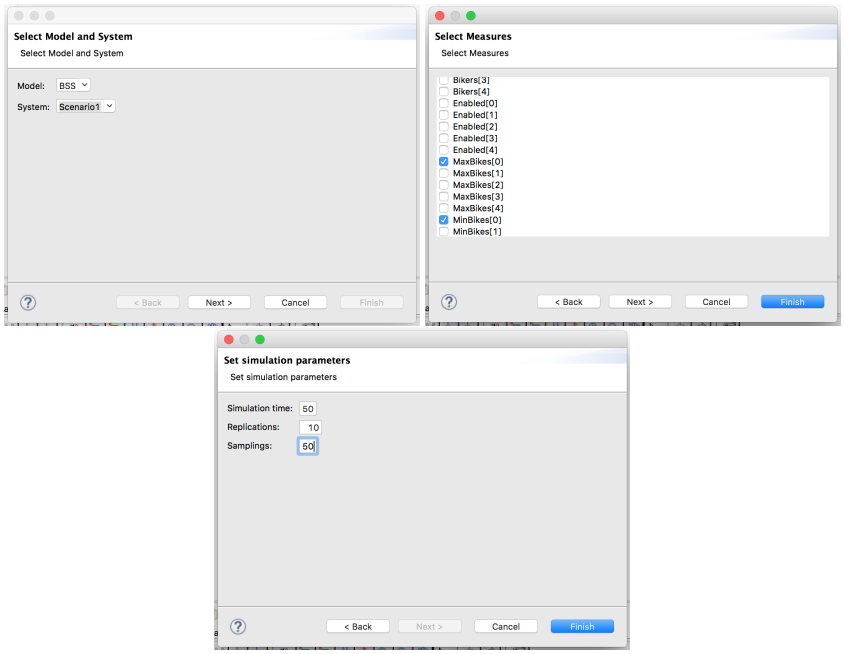

Fig. 2: CARMA Eclipse Plug-In: Simulation Wizard.

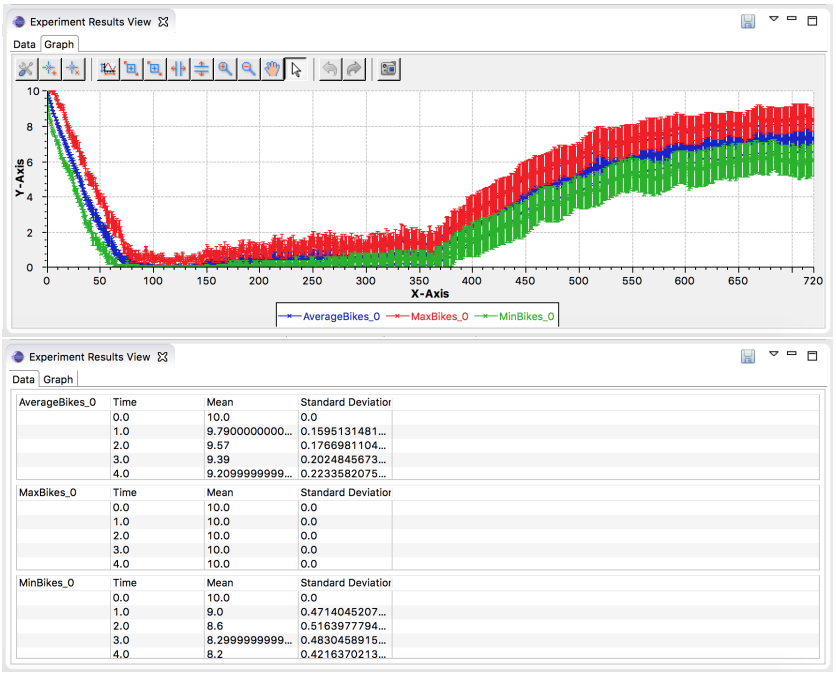

Fig. 3: CARMA Eclipse Plug-In: Experiment Results View.

```
      }
      new Arrival(i);
    }
  }
  environment {
    store {
      attrib users := 0;
    }
    prob {
      default: 1;
    }
    weight{
      [receiver.loc==sender.loc] get: 1;
      [receiver.loc==sender.loc] ret: 1;
      default: 0;
    }
    rate {
      get: get_rate;
      ret: ret_rate;
      move*: move_rate;
      arrival*: (global.users<TOTAL_USERS?arrival_rate:0.0);
      default: 1;
    }
    update {
      arrival*:
        users:=global.users+1,
        new User(sender.loc,U[0:ZONES-1]);
      ret:
        users:=global.users-1;
    }
  }
}
```

The second scenario can be simply obtained by changing the `rate` block as follows:

```
    rate {
      get: get_rate;
      ret: ret_rate;
      move*: move_rate;
      arrival*:
        (global.users<TOTAL_USERS?
          (now<360?4*arrival_rate:arrival_rate/2):0.0);
      default: 1;
    }
```

The results of the simulation of the two CARMA models are reported in Figure 4 where we report max/average/min number of bikes available at zone 0. Due to the symmetry of the considered model, any other location in the border presents similar results.

We can notice that, in both the scenarios the use of stations is not well balanced. Indeed, when the system is not overloaded, there are stations that are almost empty while others are full. This is due to the fact that stations do not collaborate and *concur*

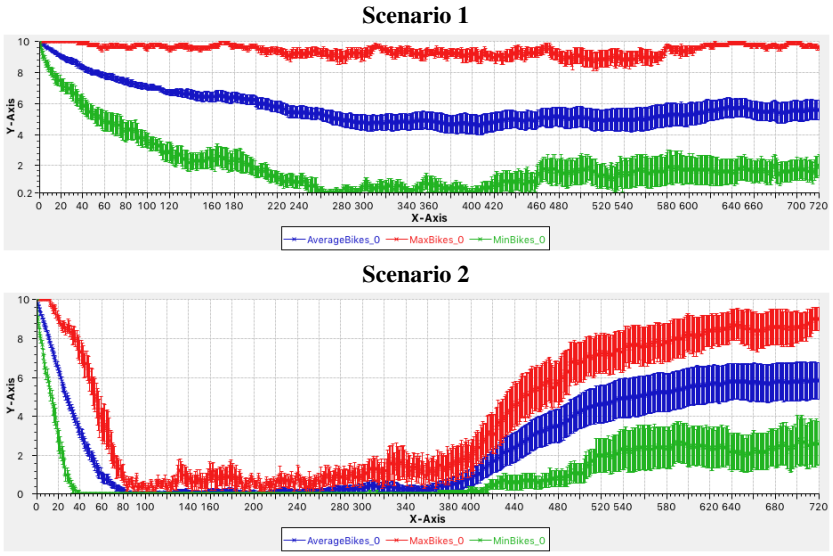

Fig. 4: Bike Sharing System: Simulation Results — 10 *simulation runs*

to attract users. To overcome this problem we change the behaviour of stations to let them exchange information about their availability. The new prototype is the following:

```
component CollaborativeStation( int loc , int capacity , int
    available ) {
  store {
    attrib loc := loc;
    attrib available := available;
    attrib capacity := capacity;
    attrib get_enabled := true;
    attrib ret_enabled := true;
  }

  behaviour {
    G = [my.available>0 && my.get_enabled]
        get<>{ my.available := my.available-1 }.G;
    R = [my.available<my.capacity && my.ret_enabled]
        ret<>{ my.available := my.available+1 }.R;
    C =
      [my.get_enabled || my.ret_enabled] spread*< my.
          available >.C
      +
      spread*[true]( x )
          { my.get_enabled := my.available >= x , my.
              ret_enabled := my.available <= x }.C;
  }
```

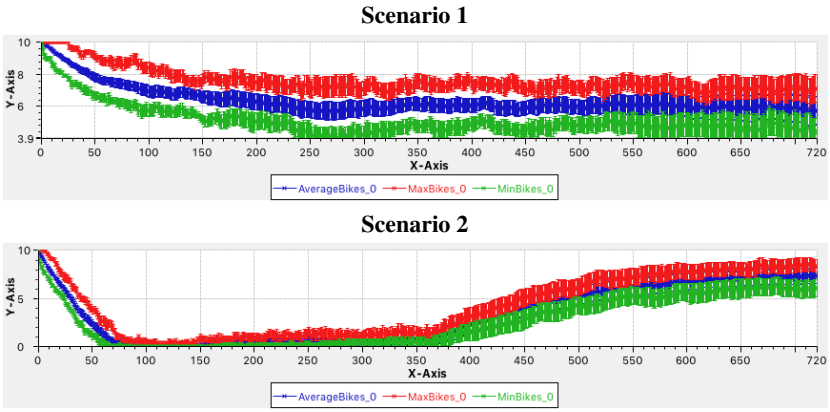

Fig. 5: Bike Sharing System (Collaborative Stations): Simulation Results — 10 *simulation runs*

```
    init {
      G|R|C
    }

}
```

`CollaborativeStations` use action `spread*` to communicate to components in the same zone the number of bikes locally available. Actions `get` and `ret`, used by users to get and return a bike, are enabled only when no other components with an higher number of bikes/parking slots is present in the zone. The simulation of these collectives in the two scenarios is reported in Figure 5. We can notice that in both the scenarios the average number of available bikes is the same as in Figure 4. However, differently from in Figure 4, the use of bikes in the stations is more balanced.

6 Concluding remarks

In this paper we have presented CARMA, a novel modelling language which aims to represent collectives of agents working in a specified environment and support the analysis of quantitative aspects of their behaviour such as performance, availability and dependability. CARMA is a stochastic process algebra-based language combining several innovative features such as the separation of behaviour and knowledge, locally synchronous and globally asynchronous communication, attribute-defined interaction and a distinct environment which can be changed independently of the agents. We have demonstrated the use of CARMA on a simple example, showing the ease with which the same system can be studied under different contexts or environments.

Together with the modelling language presented as a stochastic process algebra, we have also described a high level language (named the CARMA Specification Language) that can be used as a front-end to support the design of CARMA models and to support

quantitative analyses that, currently, are performed via simulation. To support simulation of CARMA models a prototype simulator has been also developed. This simulator, which has been implemented in Java, can be used to perform stochastic simulation and can be used as the basis for implementing other analysis techniques. These tools are available in an Eclipse plug-in that has been used to specify and verify a simple scenario.

One of the main issues related with CAS is scalability. For this reason is strongly desirable to develop alternative semantics that, abstract on the precise identities of components in a system and when appropriate offer mean-field approximation [6]. We envisage providing CARMA with a fluid semantics and in general the exploitation of scalable specification and analysis techniques [25] to provide a key focus for on-going work. In this direction we refer also here to [21] where the process language ODELINDA has been proposed which provides an *asynchronous*, tuple-based, interaction paradigm for CAS. The language is equipped both with an individual-based Markovian semantics and with a population-based Markovian semantics. The latter forms the basis for a continuous, fluid-flow, semantics definition, in a way similar to [13].

Acknowledgements

This work is partially supported by the EU project QUANTICOL, 600708. The authors thank Stephen Gilmore for his helpful comments on the chapter.

References

1. Yehia Abd Alrahman, Rocco De Nicola, and Michele Loreti. On the power of attribute-based communication. *CoRR*, abs/1602.05635, 2016.
2. Yehia Abd Alrahman, Rocco De Nicola, Michele Loreti, Francesco Tiezzi, and Roberto Vigo. A calculus for attribute-based communication. In *Proceedings of SAC 2015*, pages 1840–1845. ACM, 2015.
3. Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.
4. Marco Bernardo and Roberto Gorrieri. A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time. *Theoretical Computer Science*, 202(1-2):1–54, 1998.
5. Henrik C. Bohnenkamp, Pedro R. D'Argenio, Holger Hermanns, and Joost-Pieter Katoen. MODEST: A compositional modeling formalism for hard and softly timed systems. *IEEE Trans. Software Eng.*, 32(10):812–830, 2006.
6. Luca Bortolussi and Nicolas Gast. Mean-field limits beyond ordinary differential equations. In *SFM*. Springer, 2016.
7. Luca Bortolussi, Rocco De Nicola, Vasti Galpin, Stephen Gilmore, Jane Hillston, Diego Latella, Michele Loreti, and Mieke Massink. CARMA: Collective adaptive resource-sharing markovian agents. In *Proc. of the Workshop on Quantitative Analysis of Programming Languages 2015*, volume 194 of *EPTCS*, pages 16–31, 2015.
8. Luca Bortolussi and Alberto Policriti. Hybrid dynamics of stochastic programs. *Theor. Comput. Sci.*, 411(20):2052–2077, 2010.
9. Federica Ciocchetta and Jane Hillston. Bio-PEPA: A framework for the modelling and analysis of biological systems. *Theoretical Computer Science*, 410(33):3065–3084, 2009.

10. Paola De Maio. Bike-sharing: Its history, impacts, models of provision, and future. *Journal of Public Transportation*, 12(4):41–56, 2009.
11. Rocco De Nicola, Diego Latella, Michele Loreti, and Mieke Massink. A uniform definition of stochastic process calculi. *ACM Comput. Surv.*, 46(1):5, 2013.
12. Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A formal approach to autonomic systems programming: The SCEL language. *TAAS*, 9(2):7, 2014.
13. Cheng Feng and Jane Hillston. PALOMA: A process algebra for located markovian agents. In *Quantitative Evaluation of Systems - 11th International Conference, QEST 2014, Florence, Italy, September 8-10, 2014. Proceedings*, volume 8657 of *LNCS*, pages 265–280. Springer, 2014.
14. C. Fricker and N. Gast. Incentives and redistribution in bike-sharing systems, 2013. Online; accessed 17-September-2013].
15. Vashti Galpin. Spatial representations and analysis techniques. In *SFM*. Springer, 2016.
16. Daniel T Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22(4):403 – 434, 1976.
17. Holger Hermanns, Ulrich Herzog, and Joost-Pieter Katoen. Process algebra for performance evaluation. *Theor. Comput. Sci.*, 274(1-2):43–87, 2002.
18. Holger Hermanns and Michael Rettelbach. Syntax, Semantics, Equivalences and Axioms for MTIPP. In U. Herzog and M. Rettelbach, editors, *Proc. of 2nd Process Algebra and Performance Modelling Workshop*, 1994.
19. Jane Hillston. *A Compositional Approach to Performance Modelling*. CUP, 1995.
20. Jane Hillston and Michele Loreti. Specification and analysis of open-ended systems with CARMA. In *Agent Environments for Multi-Agent Systems IV - 4th International Workshop, E4MAS 2014 - 10 Years Later, Paris, France, May 6, 2014, Revised Selected and Invited Papers*, volume 9068 of *Lecture Notes in Computer Science*, pages 95–116, 2015.
21. Diego Latella, Michele Loreti, and Mieke Massink. Investigating fluid-flow semantics of asynchronous tuple-based process languages for collective adaptive systems. In Tom Holvoet and Mirko Viroli, editors, *Prof. of COORDINATION 2015*, volume 9037 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2015.
22. Diego Latella, Michele Loreti, Mieke Massink, and Valerio Senni. Stochastically timed predicate-based communication primitives for autonomic computing. In Nathalie Bertrand and Luca Bortolussi, editors, *Proceedings Twelfth International Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2014, Grenoble, France, 12-13 April 2014.*, volume 154 of *EPTCS*, pages 1–16, 2014.
23. Corrado Priami. Stochastic π-calculus. *The Computer Journal*, 38(7):578–589, 1995.
24. Julien Saunier, Flavien Balbo, and Suzanne Pinson. A formal model of communication and context awareness in multiagent systems. *Journal of Logic, Language and Information*, 23(2):219–247, 2014.
25. Andrea Vandin and Mirco Tribastone. Quantitative abstractions for collective adaptive systems. In *SFM*. Springer, 2016.
26. Danny Weyns and Tom Holvoet. A formal model for situated multi-agent systems. *Fundam. Inform.*, 63(2-3):125–158, 2004.
27. Danny Weyns, Andrea Omicini, and James Odell. Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, 2007.